Name of Student: Pushkar Sane		
Roll Number: 45	**Assignment Number: 2**	
Title of Assignment: Python Assignment 2 (Modularization and Classes).		
DOP: 02-11-2023	**DOS: 03-11-2023**	
CO Mapped: **CO2**	**PO Mapped:** **PO5, PSO1**	**Signature:**

Python Assignment No. 2

1. **How do we implement abstract methods in python? Give an example for the same.**
   In Python, abstract methods are part of Abstract Base Classes (ABCs), which are provided by the abc module. Abstract methods are declared in abstract classes without providing an implementation in the abstract class itself. They are intended to be overridden in concrete subclasses that inherit from the abstract class.

   Theoretical Explanation:

   a. **Importing the abc Module:**
      i. Import the abc module to create and use abstract base classes.

   b. **Creating an Abstract Class:**
      i. Define an abstract class by inheriting from ABC or setting the metaclass as ABCMeta.
      ii. Declare abstract methods within the abstract class using the @abstractmethod decorator. These methods have no implementation in the abstract class but act as placeholders.

   c. **Implementing Abstract Methods:**
      i. Subclasses that inherit from the abstract class must provide concrete implementations for the abstract methods.
      ii. Concrete subclasses are required to define and implement all abstract methods to avoid runtime errors.

   Example:

```
from abc import ABC, abstractmethod
# Define an abstract class
class Shape(ABC):
    @abstractmethod
    def calculate_area(self):
        pass

# Subclass implementing the abstract class
class Square(Shape):
    def __init__(self, side):
        self.side = side
```

```
    def calculate_area(self):
        return self.side * self.side

# Subclass implementing the abstract class
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
    def calculate_area(self):
        return 3.14 * self.radius * self.radius

# Creating instances of subclasses
square = Square(5)
circle = Circle(3)
# Calling the implemented methods
print("Area of square:", square.calculate_area())  # Output: Area of square: 25
print("Area of circle:", circle.calculate_area())  # Output: Area of circle: 28.26
```

In the example:

a. Shape is the abstract class containing the abstract method calculate_area.
b. Square and Circle are concrete subclasses that inherit from Shape and provide implementations for the calculate_area method.

This demonstrates how abstract classes and methods define a blueprint, ensuring that specific methods must be implemented in subclasses. By doing this, the subclasses adhere to a particular structure or interface, maintaining consistency and ensuring all necessary methods are implemented.

2. Is function overloading supported by Python? Give reasons.

In Python, function overloading, as commonly understood in statically-typed languages such as C++ or Java, is not directly supported due to the language's dynamic and flexible nature. Function overloading typically refers to defining multiple functions with the same name but different parameter lists. The appropriate function to be called is determined based on the number and types of arguments provided during the function call.

Reasons why function overloading is not directly supported in Python:

1. **Dynamic Typing:** Python is a dynamically typed language. Unlike statically typed languages, where types are checked at compile-time, Python checks types at runtime. This dynamic nature of Python makes it challenging to distinguish between different functions based solely on the types or number of arguments.
2. **Parameter Flexibility:** Python functions support a flexible number of arguments using default parameter values, variable-length argument lists (*args and **kwargs), and keyword arguments. This flexibility eliminates the need for function overloading based solely on the number or types of arguments.
3. **No Function Signatures:** Unlike statically-typed languages, Python doesn't rely on function signatures during compilation. Python functions are dynamically defined and invoked, making it challenging to differentiate functions based solely on their signatures.
4. **Duck Typing Philosophy:** Python follows the principle of "duck typing," emphasizing an object's behavior over its type. This allows different types of objects to be treated similarly if they exhibit the necessary behavior, reducing the emphasis on type-based overloading.

Pythonic Approach:

Instead of relying on traditional function overloading, Python emphasizes achieving similar functionality through other means such as:

1. Leveraging default arguments to provide different behaviors based on the number of provided arguments.
2. Utilizing flexible variable argument handling to accommodate varying argument counts and types.
3. Implementing conditional logic within functions to adapt behavior based on the types or number of arguments received.

Python encourages a more flexible and adaptable approach to function implementation, focusing on readability and simplicity, without relying on explicit function overloading based solely on argument types or numbers.

3. **Explain the importance of self in Python classes.**

In Python, the self parameter in class methods serves as a reference to the instance of the class. It's a naming convention that provides a way for methods to access and operate on attributes or other methods associated with a specific instance. The significance of self in Python classes can be understood as follows:

1. **Instance Specificity:**
    a. Differentiation between Instances: self distinguishes one instance of a class from another, allowing each instance to maintain its own set of attributes and methods.
    b. Accessing Instance Members: It enables methods to access instance-specific attributes and methods, helping in working with instance-level data.
2. **Attribute and Method Access:**
    a. Accessing Attributes: Through self, instance attributes can be accessed, modified, or created within methods. It points to the current instance's attributes.
    b. Invoking Methods: Methods within the class are called using self, allowing access to other methods and attributes of the same instance.
3. **Object Initialization:**
    a. Initialization via __init__: In the constructor method (__init__), self is used to initialize instance-specific attributes during object creation.
4. **Clarity and Convention:**
    a. Improves Readability: By using self as the first parameter, the code becomes more readable, indicating that the method belongs to the class instance.
    b. Best Practices: The usage of self is a Python convention, promoting consistency and understanding among developers.

Example:

```
class ExampleClass:
    def __init__(self, value):
        self.value = value  # Initializing an instance attribute
    def display_value(self):
        print(f"Instance value: {self.value}")
```

```
# Creating instances and accessing attributes
instance_1 = ExampleClass(10)
instance_2 = ExampleClass(20)

instance_1.display_value()  # Output: Instance value: 10
instance_2.display_value()  # Output: Instance value: 20
```

4. **Explain the usage of keyword ‘pass’ in class definition.**

In Python, the pass keyword is used as a null statement, allowing a block of code to exist syntactically without performing any action. Within a class definition, the pass statement serves as a placeholder, indicating that a specific block, method, or class is intentionally left empty and will be implemented at a later stage. When used within class definitions, pass has several purposes:

Placeholder in Class Definition:

1. **Minimal or Empty Class Structure:**
    a. Pass is utilized to define an empty class structure when no immediate implementation details, methods, or attributes are required. It serves as a placeholder that satisfies the syntax requirement for a class without defining any additional content.
2. **Future Development or Extension:**
    a. It signifies that a class has been outlined but its actual implementation is pending or to be extended in the future. This placeholder indicates that the class is intentionally empty at the moment and will be developed later with specific methods and attributes.
3. **Creating Abstract Methods or Classes:**
    a. Pass is used within abstract classes or when defining abstract methods using the abstract method decorator from the abc module. It acts as a placeholder for methods that should be implemented in subclasses.

Reasons for Using pass in Class Definition:

1. **Syntax Requirement:** It satisfies the need for a syntactically complete class structure without immediate implementation details.
2. **Future Development Signal:** It clearly communicates that the class is deliberately empty and intended for further development or extension.

3. **Clarity and Readability:** It provides a clear marker indicating that the empty block or class is a deliberate placeholder.

Using pass in class definitions allows developers to create an outline or structure in the code, providing a placeholder for future development or implementation. It's particularly helpful when structuring code where the presence of a class is required for organizational purposes, but the specific content is planned to be added or extended later.